Competitive Security Assessment

Zircuit_zrc_token

Sep 18th, 2024

Secure3

secure3.io

Summary

This report is prepared for the project to identify vulnerabilities and issues in the smart contract source code. A group of NDA covered experienced security experts have participated in the Secure3's Audit Contest to find vulnerabilities and optimizations. Secure3 team has participated in the contest process as well to provide extra auditing coverage and scrutiny of the finding submissions.

The comprehensive examination and auditing scope includes:

• Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.

• Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.

• Using static analysis tools to analyze smart contracts against common known vulnerabilities patterns.

• Verify the code base is compliant with the most up-to-date industry standards and security best practices.

• Comprehensive line-by-line manual code review of the entire codebase by industry experts.

The security assessment resulted in findings that are categorized in four severity levels: Critical, Medium, Low, Informational. For each of the findings, the report has included recommendations of fix or mitigation for security and best practices.

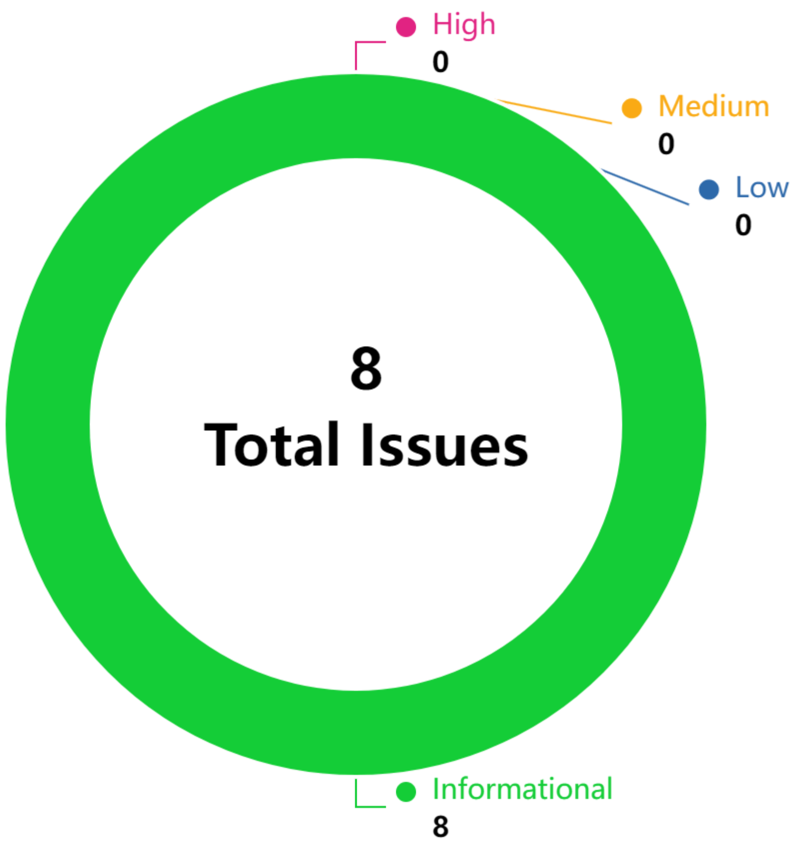
Overview

Project Name	Zircuit_zrc_token
Language	solidity
Codebase	• https://github.com/zircuit-labs/zrc-token • audit version-37cc78f8f66b646adf847530919023bcca2cd8ef • final version-37cc78f8f66b646adf847530919023bcca2cd8ef

Audit Scope

File	SHA256 Hash
src/ZRCL2.sol	d5c714de4b4916879630379491 64fdbb619e2228fea71c6026dbe423b64a3c1f
src/ZRC.sol	9f2fd97fca2eb9a3bc9b9b7c3927e2e176357522c803b20a9c3f8cbee11f690c

Code Assessment Findings



ID	Name	Category	Severity	Client Response	Contributor
ZZT-1	Use calldata instead of memory	Gas Optimization	Informational	Acknowledged	***
ZZT-2	Two-step ownership transfer	Logical	Informational	Acknowledged	***
ZZT-3	Once the contract is unlocked, it cannot be re-locked.	Logical	Informational	Acknowledged	***
ZZT-4	Missing events	Code Style	Informational	Acknowledged	***
ZZT-5	Lack Check of Zero Address	Logical	Informational	Acknowledged	***
ZZT-6	Increments can be unchecked	Gas Optimization	Informational	Acknowledged	***
ZZT-7	Crosschain transferring from Zircuit chain to ETH chain can lead to ZRC stuck in the bridge	DOS	Informational	Mitigated	***
ZZT-8	Cache array length out of the loop to save gas	Gas Optimization	Informational	Acknowledged	***

ZZT-1:Use calldata instead of memory

Category	Severity	Client Response	Contributor
Gas Optimization	Informational	Acknowledged	***

Code Reference

- code/src/ZRC.sol#L86-L103

```
86: function setAllowedSenders(address[] memory _allowedSenders, bool allow) external onlyOwner {
87:         for (uint256 i = 0; i < _allowedSenders.length; ++i) {
88:             allowedSenders[_allowedSenders[i]] = allow;
89:             emit SetAllowedSenders(_allowedSenders[i], allow);
90:         }
91:     }
92:
93:     /**
94:      * @notice set a list of receivers to be allowed or disallowed to receive tokens during the locked period
95:      * @param _allowedReceivers List of addresses to allow transfer to
96:      * @param allow Boolean to set if the receivers are allowed or disallowed
97:      */
98:     function setAllowedReceivers(address[] memory _allowedReceivers, bool allow) external onlyOwner {
99:         for (uint256 i = 0; i < _allowedReceivers.length; ++i) {
100:             allowedReceivers[_allowedReceivers[i]] = allow;
101:             emit SetAllowedReceivers(_allowedReceivers[i], allow);
102:         }
103:     }
```

- code/src/ZRCL2.sol#L156-L173

```
156: function setAllowedSenders(address[] memory _allowedSenders, bool allow) external onlyOwner {
157:         for (uint256 i = 0; i < _allowedSenders.length; ++i) {
158:             allowedSenders[_allowedSenders[i]] = allow;
159:             emit SetAllowedSenders(_allowedSenders[i], allow);
160:         }
161:     }
162:
163:     /**
164:      * @notice set a list of receivers to be allowed or disallowed to receive tokens during the locked period
165:      * @param _allowedReceivers List of addresses to set allowedReceivers
166:      * @param allow Boolean to set if the receivers are allowed or disallowed
167:      */
168:     function setAllowedReceivers(address[] memory _allowedReceivers, bool allow) external onlyOwner {
169:         for (uint256 i = 0; i < _allowedReceivers.length; ++i) {
170:             allowedReceivers[_allowedReceivers[i]] = allow;
171:             emit SetAllowedReceivers(_allowedReceivers[i], allow);
172:         }
173:     }
```

Description

***: Using `calldata` instead of `memory` for array parameters can save gas if the array contains large amounts of elements.

Recommendation

***: Consider following fix:

```
function setAllowedSenders(address[] calldata _allowedSenders, bool allow) external onlyOwner
```

```
function setAllowedReceivers(address[] calldata _allowedReceivers, bool allow) external onlyOwner
```

Client Response

client response : Acknowledged.
We acknowledge the optimization suggestion. However, since the contract has been deployed and the issue is only of informational severity, it does not make sense to fix the issue.

ZZT-2:Two-step ownership transfer

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	***

Code Reference

- code/src/ZRC.sol#L9

```
9: contract ZRC is Ownable, ERC20Permit {
```

Description

***: `Ownable2Step.sol` is safer than `Ownable.sol` for smart contracts because the owner cannot accidentally transfer smart contract ownership to a mistyped address. Rather than directly transferring to the new owner, the transfer only completes when the new owner accepts ownership.
Check the docs and the code here.

Recommendation

***: The OpenZeppelin's `Ownable2Step.sol` provides added safety due to its securely designed two-step process. Consider using `Ownable2Step.sol` instead of `Ownable.sol`.

Client Response

client response : Acknowledged.
We acknowledge the good suggestion made in this issue. However, since the contract has been deployed already and this is an informational severity issue, we will not be fixing it. It's worth mentioning that the owner role will only be relevant until the token transfers have been unlocked. At that point the owner role will be worthless and can be renounced.

ZZT-3:Once the contract is unlocked, it cannot be re-locked.

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	***

Code Reference

- code/src/ZRC.sol#L36
- code/src/ZRC.sol#L75-L79

```
36: locked = true;
```

```
75: function unlock() external onlyOwner {
76:         require(locked, "Transfer already unlocked");
77:         locked = false;
78:         emit Unlocked();
79:     }
```

- code/src/ZRCL2.sol#L48
- code/src/ZRCL2.sol#L145-L149

```
48: locked = true;
```

```
145: function unlock() external onlyOwner {
146:         require(locked, "Transfer already unlocked");
147:         locked = false;
148:         emit Unlocked();
149:     }
```

Description

***: The contract is deployed with `locked = true` and if the owner calls `unlock()` then the state of the `locked = false`, which means the state is changed to false and the tokens are unlocked, the problem is there is no way to lock them again , In case tokens are need to be locked again there is no way of doing it again since there's no `lock()` function.

Recommendation

***: Add a lock() function:

```
function lock() external onlyOwner {
    require(!locked, "Transfer already locked");
    locked = true;
    emit Locked();
}
```

Client Response

client response : Acknowledged.
This behavior is designed to limit centralization. We didn't want it to be possible to arbitrarily freeze the transfers again in the future after the initial unlock.

ZZT-4:Missing events

Category	Severity	Client Response	Contributor
Code Style	Informational	Acknowledged	***

Code Reference

- code/src/ZRCL2.sol#L42-L55

```
42: constructor(
43:         address[] memory _allowedSenders,
44:         address[] memory _allowedReceivers,
45:         address _bridge,
46:         address _remoteToken
47:     ) ERC20("Zircuit", "ZRC") Ownable(msg.sender) ERC20Permit("Zircuit") {
48:         locked = true;
49:
50:         allowedSenders[msg.sender] = true;
51:         emit SetAllowedSenders(msg.sender, true);
52:
53:         BRIDGE = _bridge;
54:
55:         REMOTE_TOKEN = _remoteToken;
```

Description

***: In the contract `ZRCL2.sol`, `constructor` sets the `BRIDGE` and `REMOTE_TOKEN` addresses, which are the key addresses for admin and users. It is a better practice to emit the corresponding event for transparency and readability.

Recommendation

***: Consider that emit corresponding events after setting key addresses.

Client Response

client response : Acknowledged.
Since the contract has already been deployed, this issue is no longer relevant.

ZZT-5:Lack Check of Zero Address

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	***

Code Reference

- code/src/ZRC.sol#L31-L51

```
constructor(address[] memory _allowedSenders, address[] memory _allowedReceivers, uint256 totalSupply)
        ERC20("Zircuit", "ZRC")
        Ownable(msg.sender)
        ERC20Permit("Zircuit")
    {
        locked = true;
        allowedSenders[msg.sender] = true;
        emit SetAllowedSenders(msg.sender, true);

        for (uint256 i = 0; i < _allowedSenders.length; ++i) {
            allowedSenders[_allowedSenders[i]] = true;
            emit SetAllowedSenders(_allowedSenders[i], true);
        }

        for (uint256 i = 0; i < _allowedReceivers.length; ++i) {
            allowedReceivers[_allowedReceivers[i]] = true;
            emit SetAllowedReceivers(_allowedReceivers[i], true);
        }

        _mint(msg.sender, totalSupply);
```

```
    }
```

- code/src/ZRCL2.sol#L42-L66

```
constructor(
        address[] memory _allowedSenders,
        address[] memory _allowedReceivers,
        address _bridge,
        address _remoteToken
    ) ERC20("Zircuit", "ZRC") Ownable(msg.sender) ERC20Permit("Zircuit") {
        locked = true;

        allowedSenders[msg.sender] = true;
        emit SetAllowedSenders(msg.sender, true);

        BRIDGE = _bridge;

        REMOTE_TOKEN = _remoteToken;

        for (uint256 i = 0; i < _allowedSenders.length; ++i) {
            allowedSenders[_allowedSenders[i]] = true;
            emit SetAllowedSenders(_allowedSenders[i], true);
        }

```

```
        for (uint256 i = 0; i < _allowedReceivers.length; ++i) {
            allowedReceivers[_allowedReceivers[i]] = true;
            emit SetAllowedReceivers(_allowedReceivers[i], true);
        }
    }
```

Description

***: The constructor doesn't validate that the bridge and remote token addresses are non-zero. If the contract is deployed with a zero address for the bridge, it would permanently set an invalid bridge address, potentially breaking cross-chain functionality.

```
constructor(
        address[] memory _allowedSenders,
        address[] memory _allowedReceivers,
        address _bridge,
        address _remoteToken
    ) ERC20("Zircuit", "ZRC") Ownable(msg.sender) ERC20Permit("Zircuit") {
        locked = true;

        allowedSenders[msg.sender] = true;
        emit SetAllowedSenders(msg.sender, true);

        BRIDGE = _bridge;

        REMOTE_TOKEN = _remoteToken;

        for (uint256 i = 0; i < _allowedSenders.length; ++i) {
            allowedSenders[_allowedSenders[i]] = true;
            emit SetAllowedSenders(_allowedSenders[i], true);
        }

        for (uint256 i = 0; i < _allowedReceivers.length; ++i) {
            allowedReceivers[_allowedReceivers[i]] = true;
            emit SetAllowedReceivers(_allowedReceivers[i], true);
        }
    }
```

***: The zero address is a special address that represents an uninitialized or burn address. It is represented as 0x0 or 0x0000000000000000000000000000. It has no private key, so any token deposited into the address cannot be recovered.
In the contracts `ZRC.sol` and `ZRCL2.sol`, it lacks of zero-address validation for senders and receivers addresses in the `constructor`.

Recommendation

***: add a check in the constructor

```
+ require(_bridge != address(0), "Bridge address cannot be zero");
+ require(_remoteToken != address(0), "Remote token address cannot be zero");
```

***: Check Zero addresses:

```
constructor(address[] memory _allowedSenders, address[] memory _allowedReceivers, uint256 totalSupply)
        ERC20("Zircuit", "ZRC")
        Ownable(msg.sender)
        ERC20Permit("Zircuit")
    {
        locked = true;
        allowedSenders[msg.sender] = true;
        emit SetAllowedSenders(msg.sender, true);

        for (uint256 i = 0; i < _allowedSenders.length; ++i) {
+           require(zrcAddress != address(0), "Invalid Address");
            allowedSenders[_allowedSenders[i]] = true;
            emit SetAllowedSenders(_allowedSenders[i], true);
        }

        for (uint256 i = 0; i < _allowedReceivers.length; ++i) {
+           require(zrcAddress != address(0), "Invalid Address");
            allowedReceivers[_allowedReceivers[i]] = true;
            emit SetAllowedReceivers(_allowedReceivers[i], true);
        }
```

Client Response

client response : Acknowledged.
Since the contract has already been deployed with the correct constructor parameters, this issue is no longer relevant.

ZZT-6:Increments can be unchecked

Category	Severity	Client Response	Contributor
Gas Optimization	Informational	Acknowledged	***

Code Reference

- code/src/ZRC.sol#L40-L48
- code/src/ZRC.sol#L87-L101

```
40: for (uint256 i = 0; i < _allowedSenders.length; ++i) {
41:             allowedSenders[_allowedSenders[i]] = true;
42:             emit SetAllowedSenders(_allowedSenders[i], true);
43:         }
44:
45:         for (uint256 i = 0; i < _allowedReceivers.length; ++i) {
46:             allowedReceivers[_allowedReceivers[i]] = true;
47:             emit SetAllowedReceivers(_allowedReceivers[i], true);
48:         }
```

```
87: for (uint256 i = 0; i < _allowedSenders.length; ++i) {
88:             allowedSenders[_allowedSenders[i]] = allow;
89:             emit SetAllowedSenders(_allowedSenders[i], allow);
90:         }
91:     }
92:
93:     /**
94:      * @notice set a list of receivers to be allowed or disallowed to receive tokens during the locked period
95:      * @param _allowedReceivers List of addresses to allow transfer to
96:      * @param allow Boolean to set if the receivers are allowed or disallowed
97:      */
98:     function setAllowedReceivers(address[] memory _allowedReceivers, bool allow) external onlyOwner {
99:         for (uint256 i = 0; i < _allowedReceivers.length; ++i) {
100:             allowedReceivers[_allowedReceivers[i]] = allow;
101:             emit SetAllowedReceivers(_allowedReceivers[i], allow);
```

- code/src/ZRCL2.sol#L57-L65
- code/src/ZRCL2.sol#L157-L172

```
57: for (uint256 i = 0; i < _allowedSenders.length; ++i) {
58:             allowedSenders[_allowedSenders[i]] = true;
59:             emit SetAllowedSenders(_allowedSenders[i], true);
60:         }
61:
62:         for (uint256 i = 0; i < _allowedReceivers.length; ++i) {
63:             allowedReceivers[_allowedReceivers[i]] = true;
64:             emit SetAllowedReceivers(_allowedReceivers[i], true);
65:         }
```

```
for (uint256 i = 0; i < _allowedSenders.length; ++i) {
            allowedSenders[_allowedSenders[i]] = allow;
            emit SetAllowedSenders(_allowedSenders[i], allow);
        }
    }

    /**
     * @notice set a list of receivers to be allowed or disallowed to receive tokens during the locked period
     * @param _allowedReceivers List of addresses to set allowedReceivers
     * @param allow Boolean to set if the receivers are allowed or disallowed
     */
    function setAllowedReceivers(address[] memory _allowedReceivers, bool allow) external onlyOwner {
        for (uint256 i = 0; i < _allowedReceivers.length; ++i) {
            allowedReceivers[_allowedReceivers[i]] = allow;
            emit SetAllowedReceivers(_allowedReceivers[i], allow);
        }
```

Description

***: In Solidity 0.8+, there's a default overflow check on unsigned integers. It's possible to uncheck this in for-loops and save some gas at each iteration, but at the cost of some code readability, as this uncheck cannot be made inline.The risk of overflow is inexistant for a uint256 here.

Recommendation

***: change it to

```
function setAllowedSenders(address[] memory _allowedSenders, bool allow) external onlyOwner {
    - for (uint256 i = 0; i < _allowedSenders.length; ++i) {
    + for (uint256 i = 0; i < _allowedSenders.length; ) {
        allowedSenders[_allowedSenders[i]] = allow;
        + unchecked { ++i; }
        emit SetAllowedSenders(_allowedSenders[i], allow);
    }
}
```

Client Response

client response : Acknowledged.
We acknowledge the optimization suggestion. However, since the contract has been deployed and the issue is only of informational severity, it does not make sense to fix the issue.

ZZT-7:Crosschain transferring from Zircuit chain to ETH chain can lead to ZRC stuck in the bridge

Category	Severity	Client Response	Contributor
DOS	Informational	Mitigated	***

Code Reference

- code/src/ZRC.sol#L66

```
66: !locked || from == address(0) || allowedSenders[from] || allowedReceivers[to], "Token transfer is locked"
```

Description

***: In ZRC and ZRCL2 token, there's a whitelist feature:

```
    function _update(address from, address to, uint256 amount) internal override {
        require(
            !locked || from == address(0) || allowedSenders[from] || allowedReceivers[to], "Token transfer i
s locked"
        );
        super._update(from, to, amount);
    }
```

This basically gatekeeps only transferring tokens from allowed senders to allowed receivers, except when minting
One of the critical features of ZRC and ZRCL2 tokens is the ability to cross-chain transfer back and forth. Combining these features with the whitelist feature, we will have edge cases that lead to tokens stuck forever in the L1 bridge. Here's the scenario:

1. User A is an allowed sender and allowed receiver in L2
2. User A attempts crosschain transferring from L2 to L1, with the receiver being user B (which is not an allowed receiver in L1)
3. Crosschain transferring in L2 is perfectly fine, User A's ZRCL2 token got burned correctly
4. In L1, the L1bridge is trying to transfer ZRC to user B but will get reverted because user B is not an allowed receiver

Recommendation

***: There's no easy fix for the issue since it relies heavily in the protocol's business logic. Some recommendation:

- Not allowed crosschain transferring when `locked = false`
- Make the allowed receivers and senders lists identical in both L1 and L2, then check the allowed receiver when bridging in the bridge contract

Client Response

client response : Mitigated. We will resolve the issue off-chain by making sure that the receivers and senders are always set correctly in both L2 and L1

ZZT-8:Cache array length out of the loop to save gas

Category	Severity	Client Response	Contributor
Gas Optimization	Informational	Acknowledged	***

Code Reference

- code/src/ZRC.sol#L40
- code/src/ZRC.sol#L45
- code/src/ZRC.sol#L87
- code/src/ZRC.sol#L99

```
40: for (uint256 i = 0; i < _allowedSenders.length; ++i) {
```

```
45: for (uint256 i = 0; i < _allowedReceivers.length; ++i) {
```

```
87: for (uint256 i = 0; i < _allowedSenders.length; ++i) {
```

```
99: for (uint256 i = 0; i < _allowedReceivers.length; ++i) {
```

Description

***: Reading array length at each iteration of the loop takes 6 gas (3 gas for `mload` and 3 gas to place `memory _offset`) in the stack.
Caching the array length in the stack saves around 3 gas per iteration.

Recommendation

***: It is recommended to store the array's length in a variable before the for-loop.
For instance:

```
 uint256 len = _allowedSenders.length;
 for (uint256 i = 0; i < len;) {
   ...
 }
}
```

Client Response

client response : Acknowledged.
We acknowledge the optimization suggestion. However, since the contract has been deployed and the issue is only of informational severity, it does not make sense to fix the issue.

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Invoices, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Invoice. This report provided in connection with the services set forth in the Invoices shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Invoice. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without Secure3's prior written consent in each instance.

This report is not an "endorsement" or "disapproval" of any particular project or team. This report is not an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Secure3 to perform a security assessment. This report does not provide any warranty or guarantee of free of bug of codes analyzed, nor do they provide any indication of the technologies, business model or legal compliancy.

This report should not be used in any way to make decisions around investment or involvement with any particular project. Instead, it represents an extensive assessing process intending to help our customers increase the quality of their code and high-level consistency of implementation and business model, while reducing the risk presented by cryptographic tokens and blockchain technology.

Secure3's position on the final decisions over blockchain technologies and corresponding associated transactions is that each company and individual are responsible for their own due diligence and continuous security.

The assessment services provided by Secure3 is subject to dependencies and under continuing development. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.